THE REVISION, EVALUATION, AND EXTENSION OF AN IMAGE FILE FORMAT

by

Mark A. Roman

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF SCIENCE at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1987

Copyright © 1987 Mark A. Roman

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly copies of this thesis document in whole or in part.

Signature of Author

Department of Electrical Engineering and Computer Science

June 1, 1987

Certified by

Steven L. Wertheim, Ph.D.
Thesis Supervisor

Accepted by

Professor Leonard A. Gould
Chairman, Undergraduate Thesis Committee
THE REVISION, EVALUATION, AND EXTENSION OF AN IMAGE FILE FORMAT

by

Mark A. Roman

Submitted to the Department of
Electrical Engineering and Computer Science on June 1, 1987
in partial fulfillment of the requirements for the degree of Bachelor of Science.

Abstract

The X Window System environment at MIT's Project Athena allows the creation of sophisticated user interfaces in application programs. There does not exist, however, a set of image processing tools that operate in this environment. The IFF Library attempts to provide a standard input/output library and a corresponding storage format. The IFF Library, while still under development, is revised and subsequently evaluated for use as an image input/output standard in the X Window environment.
Dedication

I would like to dedicate this work to my parents for taking the time to care, especially my father for not knowing how to give up; your love and persistence are matched only by your humility. I would also like to dedicate this work to Nu Delta Fraternity, because you were always there when I needed you.
# Table of Contents

Abstract 2  
Dedication 3  
Table of Contents 4  
List of Figures 5  

1. Introduction 6  
   1.1 Images in Athena Environment 6  
      1.1.1 Use of Images 6  
      1.1.2 X Window System 7  
   1.2 Design and Creation of the IFF 7  
      1.2.1 Creation 7  
      1.2.2 Design Goals 8  
   1.3 Image Terminology 9  

2. Design of IFF Library 11  
   2.1 Data Representation 12  
      2.1.1 Image Representation 12  
      2.1.2 Channel Representation 13  
   2.2 Channel Semantics: Channel Types 13  
   2.3 Function Library 15  
      2.3.1 Entry Points into the Library 15  
      2.3.2 Subsidiary Internal Functions 16  
      2.3.3 Auxiliary Definitions 17  

3. Application of IFF Library 18  
   3.1 Application Design Goals 18  
   3.2 Processing and Storage 19  
   3.3 Retrieval and Display 20  

4. Revisions and Additions to IFF 22  
   4.1 Library Extensions 22  
   4.2 Other Revisions 23  

5. Conclusions and Future Developments 25  
   5.1 Future Extensions 25  
   5.2 Usability of IFF 25  

Appendix A. Applications Code 27  
Appendix B. IFF Library Specification 34
List of Figures

**Figure 2-1:** Conceptual structure of Image File  
**Figure 3-1:** Conversion and Storage Process  
**Figure 3-2:** Providing Information to X
Chapter 1

Introduction

The Image File Format Library is a general purpose image storage format and function library for manipulating these stored images. It was created to provide a storage format standard which could be manipulated simply and efficiently by a variety of applications. The library provides a simple applications interface with the power to manipulate the various aspects of the image. The storage format is complete but minimal. This is necessary to keep the format as non-restrictive as possible.

The IFF Library is designed to consistently associate various blocks of data concerning the image. These blocks of homogeneous data can be quite different depending on the intended use of the information within the image. The IFF Library has defined in it several types of data blocks that are likely to be encountered, but has in no way restricted the user of the library. Blocks of user-defined data types may be manipulated without any degradation of performance. The section on Design and Implementation of the library discusses this in more detail. This flexibility and generality keeps the basic routines relatively simple. Added functionality is not necessary to manipulate the image data more effectively, but can be easily implemented given the extensible design of the Library. The Library is a set of low level routines which serve as a base for the development of higher level tools.

1.1 Images in Athena Environment

1.1.1 Use of Images

In the Project Athena environment, developers are defining the role of images in their educational software development. The images are "natural images." That is, they are a digital representation, a recording of some real object. Working with such images and
incorporating then into the environment presents certain problems which conventional computer graphics do not.

Conventional computer graphics are usually created by software specifically for that purpose. A great portion of these tools are advanced plotting and modeling tools. Natural images differ in that they are created via a digitizing camera or other such device, and are treated as a body of stored data.

Therefore, efforts are being made to create within the Athena environment development tools for manipulating images. The first step in creating these tools must be the establishment of a storage standard for the images and a corresponding library of utilities to effect input and output. The establishment of such a model and library require that the model be kept as general and extensible as possible to accommodate future requirements.

1.1.2 X Window System

The X Window system is device independent, low-level library for controlling high resolution displays supporting multiple overlapping windows. [Scheifler 86] Project Athena and the Lab for Computer Science at MIT have developed this window system, and it has been found to be an extremely powerful and flexible tool for using high resolution displays.

1.2 Design and Creation of the IFF

The Image File Format Library, or IFF, is the storage standard which holds promise for being a widely distributed storage standard for images. Critical evaluation of the design and scope of the IFF would allow a determination of its suitability and potential acceptance.

1.2.1 Creation

While somewhat misleading, the name of the library, IFF, stands for Image File Format, a term which refers to data representations of images in general, not just to this
specific example. The IFF Library was created primarily by Brian Kelleher in association with Dave Bidun and Mike Brown, of Digital Equipment Corporation's Western Research Laboratory, and also with Tony Della Ferra of Digital Equipment, currently at MIT's Project Athena. The motivation for designing the library was to provide a storage standard and interface through which graphics and images could be manipulated. This includes the use of three dimensional graphics and image processing, so the information relevant to a given application varies greatly. Subsequent changes and revisions have been proposed for IFF, which is still under development. These changes would allow greater compatibility across a variety of host hardware.

1.2.2 Design Goals

In the design of the image format and the IFF software library, several goals were articulated. The purpose of this library is to provide standard disk I/O for image files. To that end, there were several goals established. [Kelleher 86] They are:

1. The format must be easily understood.
2. The format must be general enough for a wide variety of applications
3. The format must be easily extended as new applications arise
4. The format must be amenable to efficient file storage
5. The format must be amenable to efficient file access

By satisfying these goals, the format will provide quick input and output functionality without imposing obtuse restrictions on client applications. This is especially true of the first three goals. If there is to be any acceptance of a library such as this, the programming interface and functional worth are paramount. To the ends of generality and extensibility, a general design is also necessary. It is critical to impose a minimal amount of policy concerning the representation, as this will allow for the most generality and fewest problems when expanding and revising the library.

The last two goals are especially critical when working on image data. Given the image size and fine resolution of current display technology (as well as trends toward even finer
image resolution) it is important that the images be stored efficiently, or any application operating on images will be restricted in the number of stored images immediately available to it. The quest for efficient storage must also be tempered with efficient access. Exotic encoding schemes may maximize storage efficiency, but incorporating such mechanism would reduce the efficiency of the library and make response time somewhat less than acceptable.

In the design of the software library, several goals were articulated [Kelleher 86] to complement those set forth for the format itself:

1. The library must not conflict with any of the file format goals
2. The library must be easy to use
3. The library should be very low level, providing access to the full functionality of the image format. Higher level libraries can be layered on top of the IFF Library

The only purpose of designing an effective storage format is to then exploit it in the implementation of the library. The construction of a library is to provide a consistent abstraction for the user. Poor library design only forces the user to circumvent the abstraction and invent functions. Such action only promotes inconsistency, and undermines widespread acceptance.

For ease of use, the IFF Library should be low level and afford latitude for the applications built on it. This is a direct corollary of the desire to keep the image file format general and extensible. Excess mechanism imposes undesirable overhead at such a low level. Efficiency and ease without loss of generality are vital.

1.3 Image Terminology

Some terminology is required to better understand the motivation for the discussions to follow.

- **Image** A digital representation of an image, considered as a rectangle. Each discernible location, or "pixel" on the screen is represented by coordinates. There is a one to one correspondence between a pixel and data concerning that pixel.

- **Pixel Data** This is the data that exists in the aforementioned one to one correspondence with each screen pixel. The semantic value of this data is unrestricted: color, object ID, etc. For example, a bitmapped image has one bit per pixel in the image.
-10-

- **Channel** This is a block of homogeneous data that represents one aspect of the pixel data for the entire image. The IFF Library is designed with the idea of multiple channels describing various facets of the entire stored image.

- **Map** A map allows a level of indirection when using color, for example. For instance, the pixel data in an image could be a description of the color at that particular location, or it could be an index into a table, where the actual color value is found. This process of indirection is called *mapping*, and the table mapped into is called the *map*. 
Chapter 2

Design of IFF Library

The IFF is based strongly on an idea of how to think of an image. The representation attempts to grasp this notion, as it is a very powerful way to constrain the problems of image manipulation and storage. In attempting to do so, the notion of the channel is introduced. A channel is a "slice" of the entire image, and contains all of the information about one aspect of the image. Essentially, it is very powerful to think of an image as a collection of these channels, each one describing an aspect. The entire collection of channels is prepended with an image header, as a kind of label for the image.

<table>
<thead>
<tr>
<th>Image Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel 0:</td>
</tr>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Map(s)</td>
</tr>
<tr>
<td>Channel 1:</td>
</tr>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Map(s)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 2-1: Conceptual Structure of Image File**

The actual storage of the image information on disk follows this conceptual structure. Notice that the image *and* each of the channels in Figure 2-1 is prepended by a header. This header contains information about the channel, and is also described below.
2.1 Data Representation

2.1.1 Image Representation

Heading the collection of channels is the image header. This header is a data structure containing information for use by applications built above the IFF Library and information for use by the Library itself to accomplish its operation. The representation is:

```c
typedef struct {
    unsigned long byteOrder;       /* always written 0x1234       */
    unsigned long magic;           /* Image library magic num */
    unsigned long majVersion;      /* Major and minor version */
    unsigned long minVersion;      /* numbers of the library */
    unsigned long width;           /* image size */
    unsigned long height;
    unsigned long nchannels;
    unsigned long aspectx, aspecty; /* pixel aspect ratio x:y */
    unsigned long bitmapBitOrder;  /* 0 = LeastSignificant */
    unsigned long bitmapScanlineUnit; /* {8, 16, or 32} */
    unsigned long scanlinePad;     /* {8, 16, or 32} */
} IFFImageHeader;

typedef struct {
    IFFImageHeader  imageHeader; /* per image header */
    IFFChannelHeader *chanHeader; /* per channel header */
} IFFImage;
```

This information is highly important, and was expanded to provide information concerning the design of the hardware which created the image. This information allows a measure of portability, as the library now has the parameters with which it can determine if any conversion of data is necessary before delivering it to the application using the image file.

In main memory, there is an additional representation. This internal image, _IFFImage_, is a representation identical to the IFFImage with addition of two pieces of state used to interface with the operating system. The method by which this internal representation is created and maintained requires that the corresponding parts of internal image type and external image type be kept identical.
2.1.2 Channel Representation

The basic idea of the image file format is an image constructed of one or more channels. Each channel contains zero, one, or three maps. The inclusion of more than one map is advantageous for certain color applications. A channel is the representation of a particular aspect of the image array, arranged in scanline order. The contents at any given location is decided by the creator of the image. Each channel has a header of information which includes a tag identifying the type of information stored in the channel, and the length of each entry in the channel. Channels, then, store pieces of information which are simultaneously associated with a given location on the screen. The representation of the channel header is:

```c
typedef struct {
    unsigned long channelType;      /* type of data */
    unsigned long encoding;          /* rle type or normal */
    unsigned long bitsPerPixel;      /* significant bits in pix */
    unsigned long channelSize;       /* in bytes */
    unsigned long mapBitsOut;        /* number of bits out of map */
    unsigned long mapsCount;         /* number of maps (0,1 or 3) */
    unsigned long mapsSize;          /* in bytes */
    unsigned long typeMasks[3];      /* for constructed types */
    unsigned long pad[6];            /* for future expansion */
} IFFChannelHeader;
```

This data structure specifies the syntax of the channel header. The information identifying the use of the channel’s pixel data is stored in this header.

2.2 Channel Semantics: Channel Types

The channel header contains a significant amount of semantic information available to the application program. The most important piece of information in that respect is the `ChannelType`, which serves as an identifying tag. Since a complete image is often comprised of several channels which correspond to components of common graphics and color models, several models have been defined in the library. They are as follows:

```c
#define IFF_CT_RED    0    /* RGB color model */
#define IFF_CT_GREEN  1
#define IFF_CT_BLUE   2
```
#define IFF_CT_CYAN 3  /* CMY color model */
#define IFF_CT_MAGENTA 4
#define IFF_CT_YELLOW 5

#define IFF_CT_HUE 6  /* HSV color model */
#define IFF_CT_SATURATION 7
#define IFF_CT_VALUE 8
#define IFF_CT_LIGHTNESS 9  /* for HLS model */

#define IFF_CT_Y 10  /* YIQ color model */
#define IFF_CT_I 11
#define IFF_CT_Q 12

#define IFF_CT_GREY 13

IFF_CT_RED, IFF_CT_GREEN, and IFF_CT_BLUE identify channels for a color image, or a grey scale image with IFF_CT_GREY. IFF_CT_RGB indicates that the channel contains indices into a single RGB color map. IFF_CT_CYAN, IFF_CT_MAGENTA, and IFF_CT_YELLOW refer to a an analogous color image using the Cyan-Magenta-Yellow color model. The third set are used to indicate the Hue Saturation/Lightness Value models, and the fourth set for the YIQ color model. In addition to these, there are three identifiers:

#define IFF_CT_Z 64  /* z buffer values */
#define IFF_CT_ALPHA 65  /* transparency, etc */
#define IFF_CT_OBJID 66  /* object id */

which are used for three dimensional graphics support. The next set of definitions are used to identify channels which contain indices into a color map, as opposed to direct image data:

#define IFF_CT_RGB 128  /* packed rgb */
#define IFF_CT_CMY 129  /* packed cmy */
#define IFF_CT_HSV 130  /* packed hsv */
#define IFF_CT_HLS 131  /* packed hls */
#define IFF_CT_YIQ 132  /* packed yiq */

#define IFF_CT_DATA 255  /* generic user data */
#define IFF_CT_RESERVED 256  /* types above this reserved for apps */

The flag IFF_CT_DATA indicates that the data is user specific and not an intrinsic part of the image. IFF_CT_RESERVED marks the beginning of custom types not defined by the library but needed by applications.
2.3 Function Library

Given the data structures that represent the image, the following functions are provided to operate on these structures. The library is primarily a set of functions layered on to the input/output interface of the UNIX operating system. Of course, included is knowledge about the structure of images. This is all the library is intended for, and this intention keeps the interface simple. Use of embedded semantic information is beyond the purpose of this library and should instead be layered on top of the IFF Library.

2.3.1 Entry Points into the Library

All of the functions operate on byte boundaries. Bit manipulation must be effected by the application using the IFF Library. The functions all treat images as pointers to the IFFImage structure. IFFOpen opens an existing image file in either read ("r") or read/write ("rw") mode. IFFOpen opens the image file, reads in all of the channel headers and initializes the data structures, returning a pointer to IFFImage structure if successful or a null pointer otherwise. IFFOpenFP opens an image in an analogous way, taking as its first parameter a file pointer instead of a file name. While any open file pointer can be used, this function was written to allow reading from "standard input" and writing to "standard output." These two "default" input/output provisions in the UNIX operating system greatly simplify the implementation of input and output by the software developer.

IFFCreate creates a new image, and operates analogously to IFFOpen.

IFFClose closes the image, closing the associated file and freeing the memory space allocated for the data structures. As a side effect, the function updates all of the headers in case any information was changed.

The input and output of actual data can be done either as a number of bytes or as a number of pixels. This is especially useful when manipulating generic data in a channel. IFFRead/IFFReadData read data from the current position within the image file. They read
the specified number of pixels or bytes, respectively. They return, as appropriate, the number of pixels or bytes successfully read, -1 otherwise. `IFFWrite/IFFWriteData` functions write in a matter analogous to their reading counterparts, returning the number of bytes written or -1 otherwise.

Moving to a specific location within the file is done by using the `IFFSeek` and `IFFSeekData` functions. Seeking can be thus be accomplished to a given pixel or byte. The `IFFSeek` routine will actually seek to the beginning of the byte in which the pixel begins. It is implemented this way because multiple pixels will be packed into each byte if the pixels are less than or equal to four bits in length.

Maps are read and written by the `IFFReadMap` and `IFFWriteMap` functions. They return the number of map cells that were read or written, or -1 otherwise.

### 2.3.2 Subsidiary Internal Functions

In addition to the library functions, the following function are defined for internal use and are not available to applications: `initImage`, `writeHeader`, `bytesToPixels`, `pixelsToBytes`, `padToWorld`, and `getTypeMasks`.

*InitImage* initializes the internal image header and allocates the spaces for the image header and channel headers. It returns a pointer to the initialized image structure if successful, a NULL pointer otherwise. The function `writeHeader` writes all of the headers in their appropriate places. These are used by the `IFFOpen` and `IFFCreate` functions, respectively, when opening an image.

Since the files associated with the image are byte addressable, there are routines provided to convert bytes to pixels and vice versa according to the semantic information in the headers. Additionally, there is a routine to calculate the padding bytes to be added on to a given number of bytes to bring it to a quad word length (i.e. a multiple of four bytes) and a function to locate the byte in which a given pixel is. These two function abstract the details of pixel packing methods into single function calls.
The `getTypedMasks` function takes the image and desired channel number, returning a pointer to the array of type masks. These masks can be used to elicit information from the pixel data. The `_IFFDumpImage` routine is provided as a debug routine to get the vital statistics on an image file. It prints to standard output and is of no value for an application.

### 2.3.3 Auxiliary Definitions

In addition to all of the functions are several macro definitions which provide semantic information from the image or channel headers. While any or all of these could have been implemented as functions none of them involves any processing that would fall prey to the problems of macro substitution, so they are implemented as preprocessor macros.

```markdown
#define version(x)
#define minVersion(x)
#define update(x)
#define width(x)
#define height(x)
#define AspectRatio(x)
```

While the sophisticated programmer might choose to directly access the various headers to obtain information, it much more desirable for understanding to maintain the level of abstraction. It was for this reason these macro definitions were added.

This is the extent of the Library's functionality. There is included in an Appendix the full interface specification for each of the library functions.
Chapter 3

Application of IFF Library

The Athena environment provides extensive graphics capabilities. These capabilities are based on the power of the X Window System developed at the Massachusetts Institute of Technology. Given this environment, the logical progression was to evaluate the suitability of the proposed IFF storage standard in the environment. Development of practical applications in the X environment were needed.

3.1 Application Design Goals

The applications had to accomplish the following tasks:
1. Take as input an array of image data.
2. Process the data into a format compatible with the X environment
3. Store this processed data in a standard format
4. Retrieve this data from storage and present it in the X environment

The image data was provided by an AT&T Targa-16 digitizing frame buffer. This equipment interfaces with the IBM Personal Computer and is controlled by the Targa Image Processing System (TIPS from Island Graphics). The TIPS software allowed the digitization of a video image at a resolution of 512 by 512 pixels. The image array consisted of pixel values fifteen bits in length, where each pixel value represents the color to be displayed at that location in the image. Previously written applications implemented the image processing algorithm. Thus, the tasks of the new applications were to store and then later retrieve the processed image data in the IFF format.
3.2 Processing and Storage

The first application of the two new applications uses the implemented conversion algorithm to convert the image array into an array and a color map. Each element of the processed image array is a one byte (8 bit) index into the color map. One byte allows 256 different entries in the color map. This allows only 256 simultaneous colors in the new image, while the input image had $2^{15} = 32768$ simultaneous colors. Thus, the conversion algorithm also uses histogram methods to determine which 256 of the 32768 colors to keep, and which of the chosen colors is closest to the each color not chosen. In this manner, from the 32768 color input image a 256 color approximation is created.

![Diagram](image)

**Figure 3-1: Conversion and Storage Process**

Once given a converted image, the application writes a standard IFF image file, containing the array of pixels and the color map. The implementation of this conversion program, named filter, is included in the Appendix. With these tasks completed, all that remained was an application which could easily read the IFF file using the IFF Library and readily provide this information to the X environment for display.
3.3 Retrieval and Display

The second of the two new applications reads the IFF image data. It initializes the X environment and receives from X an allocation for the color map. In doing so, it receives from X the indices which have been allocated. Thus, it is necessary to convert the old indices to the new ones.

This task is achieved by constraining the pixel values (i.e. the indices) to be in the range from 0 to 255 when first writing the image file. Given this constraint, the pixel value \( i \) in Figure 3-2 would indicate that color \( i \) in the stored color map would be put into index \( i \)th cell in the X environment. This amounts to a second level of indirection.

![Diagram showing the conversion of IFF pixel values to X color values.]

**Figure 3-2: Providing Information to X**

Once the indirection described in Figure 3-2 is resolved, the pixel values are provided to X and the color map is stored in the space allocated by X. The program has proven to work without any difficulty, including redrawing the screen.

Because X supports multiple overlapping windows, redrawing may be necessary if a portion of the window becomes obscured and then reexposed. Performance is very good. Operation of the displiff program takes about five seconds from invocation to complete window
display. This is particularly good considering the size (about 250 kilobytes) of the image. While an alternative implementation stores the image data in a buffer after reading it from the image file, the current implementation simply reread the image file for each redraw. The difference in performance is minimal, and the memory overhead is considerably less with the current implementation.
Chapter 4

Revisions and Additions to IFF

In addition to the evaluation and application of the IFF Library, it became evident that several modifications would need to be made in order to make the Library completely operational. While the current version would in fact operate with the given applications, difficulties would most likely be encountered if more advanced applications exploited the Library. Also, certain modifications had been proposed by personnel at DEC Western Research Laboratories [Mullis 87] which had not been implemented.

4.1 Library Extensions

As the purpose of the IFF Library is to provide a basic functionality, changes were made only when such changes simplified the interface or added needed functionality. Two changes/extensions of the library were made to these ends. The first of these is the splitting of the function IFFOpen into IFFOpen and IFFCreat. Previous versions of IFFOpen had the following specification:

```c
IFFOpen(fn, rw, width, height, nchannels, chanHead)
    char *fn; /* filename */
    char *rw; /* read/write mode */
    int width, height; /* dimensions */
    int nchannels; /* number of channels */
    IFFChannelHeader chanHead;
```

The channel header is required to initialize the channels when creating an image. This interface seemed overly complex, as too many arguments were required from an application simply wanting to open an image file. Only the first two arguments are used to open an existing image file. The remaining four arguments are used only to convey semantic information to the Library when it is creating a new image. Thus, the implementation of
IFFCreat as a separate function reduces the burden on those applications operating on existing images.

The second change was the addition of the "rw" mode. This mode allows the application to both read and write an image, as opposed to exclusively reading or exclusively writing the image. While an application could simply close and reopen an image in a different mode, this method is both cumbersome and slow, especially when the images are somewhat complex and large.

Extension of currently existing images could be achieved through new Library routines, but in the interest of simplicity, such methods are rejected in favor of creating a new image that is the combination of the old image and newly added elements.

The IFFOpenFP function was also implemented at this time. It operates as described above, and provides greater ease of programming input and output.

4.2 Other Revisions

Revisions performed on the Library were motivated by a revised data representation. As the IFF Library is still under development, there were issues raised and resolved since the last version of the library was created. Two of the tasks of this thesis were to add the suggested changes to the data representation and ensure that the Library was updated in accordance to the changes.

There were several changes suggested to the current data representation:
1. inclusion of Type Masks as part of the channel header
2. inclusion of a major and minor version number in the image header
3. inclusion of the Scanline Unit, Bit Order, and Scan Line Padding parameters in the image header

The first two modifications were made with an eye toward future dependency of applications tools built on the IFF Library. These amounted to minor modifications of the data
structures. The third modification, the inclusion of the three scanline parameters, was done to provide for future portability mechanisms. The size of each unit of data and the bit/byte order affects the ordering of individual byte in storage [Newman 87]. Thus two machines having different byte ordering and different scanline units would require conversion if the image is to shared by the two machines.

The mechanism within the reading and writing functions to accommodate the differences shown above has been implemented. That is, IFFRead and IFFWrite will perform the necessary conversions if requested. Performance is somewhat slower, but the decrease in performance is not that significant. The primary problem is the communication of the desired format to the Library upon opening an image. This can be achieved via a "site" file with this configuration information in it, although this would require recompiling the Library on each new machine. While this means the library is no longer completely portable, it is an adequate concession to make for greater image portability.
Chapter 5

Conclusions and Future Developments

5.1 Future Extensions

The Library should be extended to provide greater portability of images among machines of differing hardware designs. For true compatibility, all header information could be stored in a companion plain text file or as ASCII text within the original file. This would allow the library to easily determine the structure of the stored image data despite differences between the hardware which created the image and the hardware which is currently using it.

5.2 Usability of IFF

After this evaluation, The IFF storage model and function library have performed favorably. Applications are feasible in the X environment and are not inelegant or overly complex. Applications in version 11 of X should be a lot easier to implement than they were in version 10 because the X11 Library reflects greater consideration of graphics and color and the need for applications to effectively manipulate them [Gettys 87]. While the X Window System is a very well designed and controlled environment, it is advantageous to use a library as rigorous as IFF.

Other image manipulation tool kits can be layered on to IFF because it imposes no constraints on the image information, and supplies facilities for retaining the semantic information concerning the image. Thus, tools can identify and retrieve a channel's information, e.g. object labels in a three-dimensional simulation, and use it to interpret and manipulate the image.

IFF has a general design that constrains the problems of image manipulation very well.
The model on which IFF is conceived aids in the design of applications which manipulate images. This image storage and input/output standard would serve well as the nucleus of image manipulation tools in the Athena environment.
Appendix A

Applications Code

This is the implementation of the filter and displiff applications.

/*
 * Filter to produce X IFF files from TARGA-16 image
 * files. Written to produce GFX display - marcus 3/20/87
 * Portions of code liberally ripped off from:
 *
 * GFX display of TARGA-16 color images - JT 11/21/86
 */
#include <stdio.h>
#include <sys/file.h>
#include <X/Xlib.h>
#include "iff.h"

// Some constants */
#define NUMCOLORS 32768 /* The number of colors (i.e., 2^5 cubed ) */
#define NUMSLOTS 200 /* The number of slots in the output GFX */
#define MORE_THAN_ANYONE 500000
/* A number bigger than the biggest frequency */

#define MIN(a,b) (((a) < (b)) ? (a) : (b))
#define MAX(a,b) (((a) > (b)) ? (a) : (b))

main(argc, argv)
  int argc;
  char **argv;
{

  struct tripl { unsigned short red, green, blue;};
  struct tripl cmap[NUMSLOTS];

  char *malloc();
  short *data;
  unsigned char *image;

  int fd, open(), close();
  int height, width, xtransfer;

  register short *d;
  register unsigned char *im;

  IFFImage *outimage;
  IFFChannelReader *chanhead;

  if(argc < 2){
    fprintf(stderr,"usage: %s infile outfile\n",argv[0]);
    exit(1);
  }

  /* Read image data and process it */
  if ((fd = open(argv[1], O_RDONLY, 0)) < 0) {
    fprintf(stderr,"can't open data file") ;
    exit(1);
  }
  read_targa(fd, &width, &height, &data);
close(fd);

fprintf(stderr,"Read the targa image\n");

if((image = (unsigned char *)malloc(width*height)) == NULL) {
    fprintf(stderr,"Can't allocate image array\n");
    exit(1);
}

/* using DataCube algorithm */
conversion(image,width,height,data,cmap);

fprintf(stderr,"Made it through conversion\n");

/* this is where the IFF shit comes in 
* + the map cell size should be made 3*2 bytes + 
*/
chanhead = (IFFChannelHeader *)malloc(sizeof(IFFChannelHeader));
chanhead->bitsPerPixel = 8;
chanhead->channelSize = 0;
chanhead->mapSize = 0;
chanhead->mapBitsOut = 8*(3*sizeof(short));
chanhead->mapCount = 1;
if((outimage = IFFOpen(argv[2], "w", width, height, 1, chanhead)) == NULL) {
    fprintf(stderr,"*s : Can't open output file.\n",argv[0]);
    exit(1);
}

IFFSeekData(outimage, 0, 0);

IFFWriteData(outimage, image, (width*height));
IFFWriteMap(outimage, 0, 0, cmap, NUMSLOTS);
IFFClose(outimage);

}

struct targaheader {
    char ncharid;
    char cmaptype;
    char imagetype;
    char cmapaddress[5];
    short xorigin;
    short yorigin;
    short width;
    short height;
    char pixelsize;  // assumes 256
    char imagedesc;
} targaheader;
char imageid[256];

read_targa(fd,nx,ny,data)
int *nx, *ny, fd;
short **data;
{
    read(fd,&{targaheader.ncharid sizeof(struct targaheader)});
    if(targaheader.ncharid & 0x80) read(fd,imageid,targaheader.ncharid);
    *nx = targaheader.width;
    *ny = targaheader.height;
    if((data = (short *)malloc(2*(nx)*(ny))) == NULL) {
        fprintf(stderr,"Can't allocate input data array\n");
        exit(1);
    }
    read(fd,*data,2*(nx)*(ny));
}

/* here is conversion.c */
conversion(image, width, height, data, cmap)
char *image;
int width, height;
short *data;
struct tripl { unsigned short red, green, blue;};
struct tripl cmap[NUMSLOTS];

{
    /* Static variables for various things */
    int freq[NUMCOLORS];     /* Frequencies of the various colors */
    int most_frequent[NUMSLOTS]; /* The most frequent colors so far */
    int most_fs[NUMSLOTS];      /* and how frequent those are */
    int kick_me, searches, least_f, new_least_f;
    int planes, sq[32], distance, pixels[256], temp;

    short *d;
    char *im;

    register int i,j,k;
    Color pixel;
    fprintf(stderr,"Made it inside of conversion\n");
    for (i=0;i<NUMCOLORS;i++) freq[i]=0;

    for (j=0;j<height;j++)
       for (i=0;i<width;i++) freq[((int)((data+i+j*width))&0x7FFF)+1];

    /* OK, now the objective is to find the NUMSLOTS most popular
     * colors in there. Here's what we do: Set up a table of the
     * NUMSLOTS most popular so far. Also, note what the smallest
     * popularity in there is. Now, look at the next color. If
     * it's more than the least popular, kick out one of the colors
     * that's least popular and add the new one, and find the new
     * least popularity. I know it's a crock, but it will work
     * and I can write it in an hour. Leo */

    /* Set up first* NUMSLOTS colors as the current most popular ones */
    least_f = freq[0];
    for(i=0;i<NUMSLOTS;i++)
    {
        most_frequent[i] = i;
        most_fs[i] = freq[i];
        if (freq[i] < least_f) least_f = freq[i];
    }

    /* Now walk through the rest of the colors */
    searches = 0;
    for(i=NUMSLOTS;i<NUMCOLORS;i++)
    {
        if (freq[i] > least_f) {
            /* Then we've got to kick one of the current ones out */
            kick_me = -1;
            new_least_f = MORE_THAN_ANYONE;
            for(j=0;j<NUMSLOTS;j++)
            {
                if (not most_fs[j] < least_f && kick_me == -1) kick_me = j;
                else if (most_fs[j] < new_least_f) new_least_f = most_fs[j];
            }
            if (kick_me == -1) {
                fprintf(stderr,"Deep shit 41\n");
                exit(-1);
            }
            most_frequent[kick_me] = i;
            most_fs[kick_me] = freq[i];
            least_f = new_least_f;
            searches++;
        }
    }
}
/* Well, we know where they are... now build a color table of them. *
 * This is where the massive hack comes in. Instead of getting pix 
 * values from X, we make them up, i.e. 0<=pix<NUMSLOTS 
 */

for (i=0;i<NUMSLOTS;i++) {
    /* The aforementioned hack */
    pixels[i] = i;

    cmap[i].blue=(most_frequent[i] & 0x001F);
    cmap[i].green=(most_frequent[i] & 0x03E0)>>5;
    cmap[i].red=(most_frequent[i] & 0x7C00)>>10;
}

for (i=0;i<NUMCOLORS;i++) {
    if (freq[i]!=0) {
        pixel.blue=(i & 0x001F);
        pixel.green=(i & 0x03E0)>>5,
        pixel.red=(i & 0x7C00)>>10,
        distance=3*sq[31];
        searches=0;
        for (j=0;j<NUMSLOTS;j++) {
            kick_me=sq[abs(cmap[j].blue-pixel.blue)]+
                 sq[abs(cmap[j].green-pixel.green)]+
                 sq[abs(cmap[j].red - pixel.red)];
            if (kick_me==0) {
                searches=j;
                break;
            } else if (distance>kick_me) {
                distance=kick_me;
                searches=j;
            }
        }
        freq[i]=searches;
    }
}

fprintf(stderr,"Made it through creation of color table (conv)\n");
fprintf(stderr,"\n");
/ * ??? Targa-16 images appear to have half line offset??? */
d = data-width/2;
for (i=0;i<height;i++) {
    j = width;
    while(j--) *im++ = pixels[freq[((int)*(d++)&0x7FFF))];
    im -= 2*width;
}

fprintf(stderr,"about to finish conversion\n");
for (i=0;i<NUMSLOTS;i++) {
    cmap[i].blue=cmap[i].blue<<11;
    cmap[i].green=cmap[i].green<<11;
    cmap[i].red=cmap[i].red<<11;
}
/*
 * Display of an IFF image.
 * Written 22.4.87 marcus.
 * X option parsing ripped off.
 */

#include <stdio.h>
#include <sys/file.h>
#include <X/Xlib.h>
#include "iff.h"

/* Some constants */
#define PICTDEFAULT "+100+100"
#define NUMSLOTS 200   /* The number of slots in the output GPX */

#define MIN(a,b) (((a) < (b)) ? (a) : (b))
#define MAX(a,b) (((a) > (b)) ? (a) : (b))

main(argc, argv)
int argc;
char **argv;
{
  Window wind;
  XEvent event;
  XExposeEvent *xpev = (XExposeEvent *)event;
  char *geometry = NULL;
  char filegeometry[30]; /* created from file unless overridden */
  char *display = NULL;
  char *option;
  char *border_color, *back_color;
  OpaqueFrame Frame;
  int border_width;
  Pixmap border_pixmap;
  Color cdef;
  Color cmap[256];
  char *mcallr();

  register unsigned char *image, *im;
  register int i, j;
  register IFFImage *inImage;

  IFFChannelHeader tmpCH;
  int height, width;
  int planes, ncells, pixels[256];
  struct tripl t1 { unsigned short red, green, blue;};
  struct tripl rgb[256];

  /* Parse the command line & set up options */

  if ((option = XGetDefault(argv[0], "BorderWidth")) != NULL)
    border_width = atoi(option);
  if ((border_color = XGetDefault(argv[0], "Border")) == NULL)
    border_color = XGetDefault(argv[0], "BorderColor");
  back_color = XGetDefault(argv[0], "Background");

  for (i = 1; i < argc; i++) {
    if (argv[i][0] == '=') {
      geometry = argv[i];
      continue;
    }
    if (index(argv[i], ':') != NULL) {
      continue; /* host:display */
    }
    if (strcmp(argv[i], "-bw") == 0 ||
        strcmp(argv[i], "-border") == 0) { /* border width */
      }
if (++i >= argc) usage(aryv[0]);
    border_width = atoi(aryv[i]);
    continue;
}
if (strcmp(aryv[i], "-bd") == 0 ||
    strcmp(aryv[i], "-color") == 0) { /* border color */
    if (++i >= argc) usage(aryv[0]);
    border_color = argv[i];
    continue;
}
if (argv[i][0] == '-') usage(aryv[0]);
}

if (!XOpenDisplay(display)) {
    perror(aryv[0]);
    exit(1);
}

if (border_color && DisplayCells() > 2 &&
    XParseColor(border_color, &cdef) && XGetHardwareColor(&cdef))
    border_pixmap = XMakeTile(cdef.pixel);
else if (border_color && strcmp(border_color, "black") == 0)
    border_pixmap = BlackPixmap;
else if (border_color && strcmp(border_color, "white") == 0)
    border_pixmap = WhitePixmap;
else
    border_pixmap = BlackPixmap;

frame.bdrwidth = border_width;
frame.border = border_pixmap;
frame.background = BlackPixmap;

/* Open image, get dimensions, & allocate play space */

if((inImage = IFFOpen( argv[1], "r", 0, 0, 0, &tmpCE)) == NULL) {
    fprintf(stderr, "** can't open input image file\n", argv[0]);
    exit(1);
}

width = inImage->imageHeader.width;
height = inImage->imageHeader.height;

if((image = (unsigned char *)malloc(2 * width)) == NULL) {
    fprintf(stderr, "** can't allocate image array\n", argv[0]);
    exit(1);
}

IFFSeek(inImage, 0, 0, 0);

/* Allocate and
 * Read map into an array rgb[# of cells]
 * cast as a buffer
 */
if (XGetColorCells(0, NUMSLOTS, 0, &planes, pixels) == 0) {
    fprintf(stderr, "** Insufficient color planes resource\n", argv[0]);
    exit(1);
}

tocells = IFFReadMap(inImage, 0, 0, (unsigned char *)rgb, NUMSLOTS);
fprintf(stderr, "** bd map entries read\n", argv[0], tocells);

/* Use as an array & put into cmap -- add pixels, too */

for(i=0;i<NUMSLOTS;i++){
    /* dummy mode debug */
    cmap[i].pixel = pixels[i];
    cmap[i].red = 1<<15;
    cmap[i].green = 0;
cmap[i].blue = 0;
*/
cmap[i].pixel = pixels[i];
cmap[i].red = rgb[i].red;
cmap[i].green = rgb[i].green;
cmap[i].blue = rgb[i].blue;
}
XStoreColors(NUMSLOTS, cmap);

/* Create a window & put in data */

sprintf(filegeometry, "=dx%d\s", width, height, PICTDEFAULT);
wind = XCreate(argc[0], argv[0],
               geometry, filegeometry, &frame, 100, 100);
if (!wind) {
    fprintf(stderr, "XCreateWindow failed\n");
    exit(1);
}

XSelectInput(wind, ButtonPressed | ExposeRegion | ExposeWindow);
XMapWindow(wind);

while(1) {
    XNextEvent(&event);
    switch((int)event.type) {
    case ExposeWindow:
    case ExposeRegion:
        for(i=0;i<height;i++) {
            IFFRead(inimage, image, width);

            /* Do the conversion of pix values here */
            j = width;
            im = image;
            while(j--) {
                *im++ = (unsigned char)cmap[(int)(*im & 0xFF)].pixel;
            }
            XPixmapBitsPutE(wind, 0, i, width, 1, image,
                            0, 0, AllPlanes);
        }
        break;
    }
    XFlush();
}

usage (program)
    char *program;
    {
        fprintf(stderr,"usage: %s filename [ options ] \n", program);
        fprintf(stderr,"    where options are one or more of:\n");
        fprintf(stderr,"    [host:display] [=geom] [-bw] [-bd]\n");
        exit(1);
    }
Appendix B

IFF Library Specification

These are the interface specifications. Analogous to the low level input output library on a standard UNIX system, the IFFRead/Data and IFFWrite/Data functions return the number of pixels or bytes transferred, as appropriate, or -1 if the operation was unsuccessful. IFFReadmap and IFFWritemap return the number of map cells transferred, or -1 if unsuccessful.

The seek routines return 0 if successful, -1 otherwise. IFFOpen and IFFCreate return a pointer to an image, a null pointer if unsuccessful. All operations operation on byte boundaries.

```c
// IFFImage *IFFCreate(filename, rw, w, h, nchannels, *channelHeader)
char *filename;
char *rw;
int w, h, nchannels;
IFFChannelHeader *channelHeader;

// IFFImage *IFFOpen(filename, rw)
char *filename;
char *rw;

int IFFClose(image)
    IFFImage *image;

int IFFSeek(/* image, x, y, channel */);
    IFFImage *image;
    int x, y, channel;

int IFFSeekData(/* image, channel, chanOffset */);
    IFFImage *image;
    int channel, chanOffset;

int IFFRead(/* image, buf, npixels */);
    IFFImage *image;
    char *buf;
    int npixels;

int IFFReadData(/* image, buf, nbytes */);
    IFFImage *image;
    char *buf;
    int nbytes;

int IFFWrite(/* image, buf, npixels */);
    IFFImage *image;
    char *buf;
    int npixels;

int IFFWriteData(/* image, buf, nbytes */);
```
IFFImage *image;
char *buf;
int nbytes;

int IFFReadmap(/* image, mapnum, startindex, buf, n */);
IFFImage *image;
int mapnum, startindex;
char *buf;
int n;

int IFFWriteMap(/* image, mapnum, startindex, buf, n */);
IFFImage *image;
int mapnum, startindex;
char *buf;
int n;
References

[Kelleher 87] Kelleher, Brian M.
Personal Communication.
March, 1987

X11 C Language X Interface Protocol Version 10
MIT Project Athena, 1986.

X11 C Language X Interface Protocol Version 11

IFF: An Image File Format.

[Kelleher 87] Kelleher, Brian M.
Personal Communication.
April, 1987

[Kernighan 78] Kernighan, Brian and Ritchie, Dennis.
The C Programming Language.

[Mullis 87] Mullis, Don W.
Personal Communication.
May, 1987

Personal Communication.
April, 1987

[Scheifler 86] Scheifler, Robert W. and Gettys, Jim.
The X Windows System.

[Scheifler 87a] Scheifler, Robert W.
X Window System Protocol, Version 11

[Scheifler 87b] Scheifler, Robert W.
Personal Communication.
April, 1987